

Manuale software IRIS

Massimo Giuseppe Martello

2025

Contents

1	Introduzione al software di controllo dell'esperimento IRIS	2
2	General overview	2
2.1	Motori ed attuatori	3
2.2	Charge - Irradiation - Discharge	4
2.2.1	Charge	4
2.2.2	Irradiation	5
2.2.3	Discharge	5
3	Lasal Class 2 - Control programming	7
3.1	Motors and Actuators	7
3.2	Struttura software di controllo	9
3.3	Hardware network	9
3.4	Automatic Network	10
3.4.1	Global Manager	11
3.4.2	Main Sequence	12
3.4.3	Motors' manager	14
3.5	Manual network	20
3.6	Axis Profiles network	22
3.7	ModBus Communication network	23
4	Descrizione classi singoli motori	25
4.1	Motore M1 - Charge slider motor	25
4.2	Motore M2 - Charge buffer motor	25
4.3	Motore M3 - Discharge Slider	27
4.4	Motore M4 - Discahrge Buffer	27
4.5	Motore M5 - Central Movement	29
4.6	Motore M6 - Allineamento camera	31
4.7	Attuatore 1 & 2	32
4.8	STO - Arresto di emergenza	32

1 Introduzione al software di controllo dell'esperimento IRIS

Attraverso il presente documento si intende fornire una guida all'utilizzo dell'esperimento IRIS, installato presso l'edificio SPES, sala A13. Verranno illustrate le principali componenti coinvolte nella movimentazione dei target secondari o più semplicemente chiamata d'ora in avanti pellet.

2 General overview

L'esperimento IRIS si trova installato nell'area a bassa energia sulla linea ISOLPHARM all'interno dell'edificio SPES, figura 1.

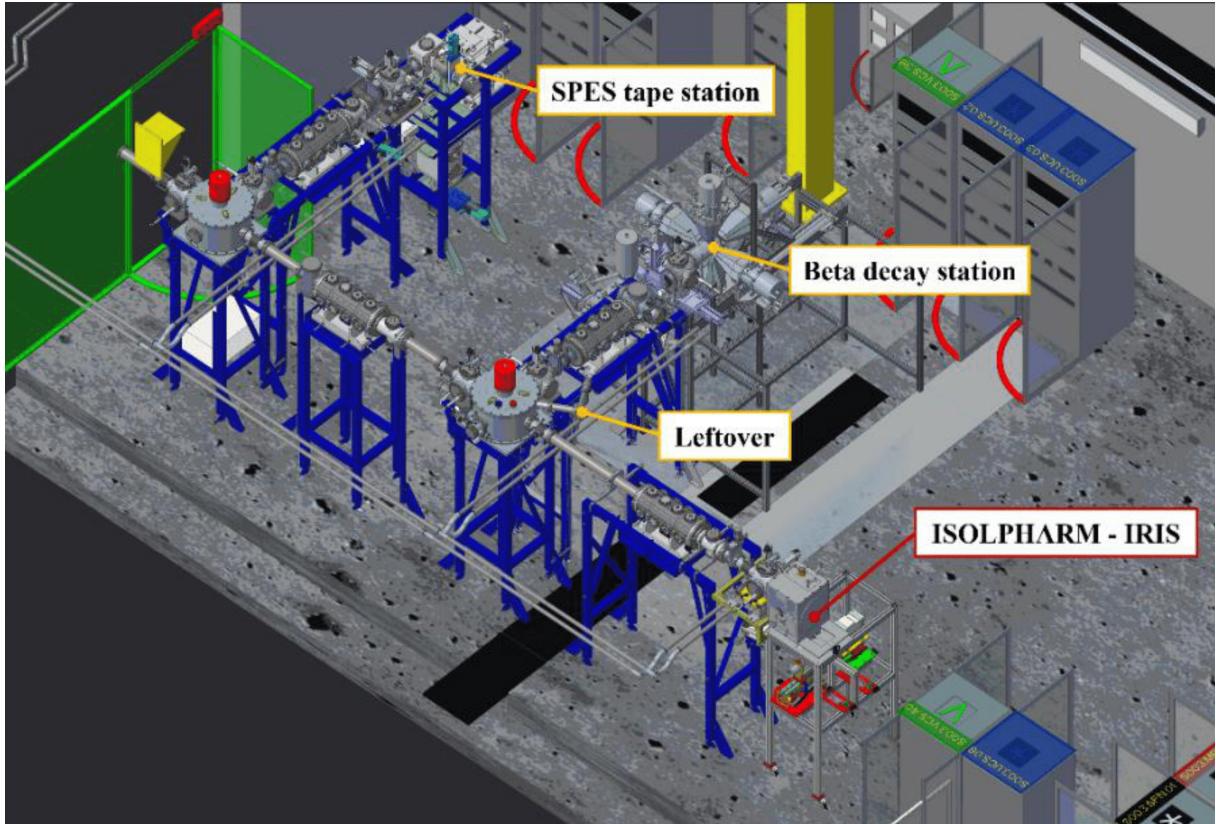


Figure 1: Pianta

Lo scopo dell'esperimento IRIS è consentire la raccolta di radionuclidi su specifiche pastiglie per applicazioni di tipo medico. L'intero processo di movimentazione delle pastiglie, dalla zona di carico, alla zona di irraggiamento, fino a quella di misurazione e scarico, è completamente automatizzato. Per rendere il sistema facilmente gestibile da un operatore, senza la necessità che quest'ultimo conosca nei dettagli il codice di controllo, sono stati implementati diversi elementi, rappresentati schematicamente in figura 2.

Gli elementi utilizzati sono:

- Control Software → software di controllo sviluppato in LASAL
- EPICS IOc → applicativo che permette la condivisione di alcune variabili dal PLC ad un'interfaccia grafica attraverso il protocollo di comunicazione Modbus.
- CS-Studio → interfaccia grafica (GUI).

L'idea di fondo è stata quella di sviluppare un software scalabile ovvero facilmente modificabile nelle funzioni già implementate ma anche facilmente espandibile con altre di nuove. L'ambiente di sviluppo è Lasal Class 2, fornito dalla casa madre dei PLC, Sigmatek.

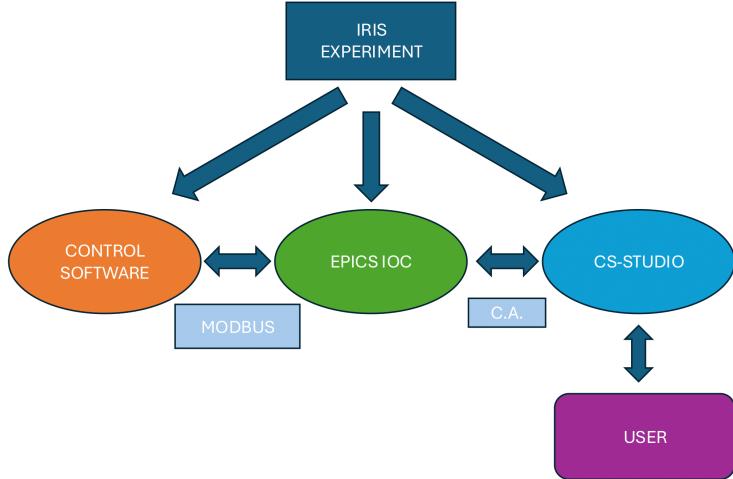


Figure 2: IRIS organization

Il software di controllo progettato consente la movimentazione dei *pellet* da una posizione iniziale di carica a una posizione finale di raccolta, attraversando posizioni intermedie mediante l'impiego di motori passo-passo e attuatori (solenoidi). Il ciclo di lavoro è suddiviso in tre macro-fasi:

- Fase di caricamento dei pellet
- Fase di irraggiamento
- Fase di scarica e misurazione del pellet

Le modalità di funzionamento previste sono due:

- Base
- Avanzata

Nella modalità base il ciclo di lavoro risulta essere più automatizzato e quindi più facilmente monitorabile e interagibile con l'utente di turno. In questa modalità l'utente dovrà inserire i parametri richiesti, ovvero: pellet totali caricati, tempo di irraggiamento per ciascun pellet ed interagire con il sistema attraverso comandi di start, inizio caricamento, inizio irraggiamento e inizio e fine misurazione. Il passaggio dalla fase di carica a quella di irraggiamento ed infine a quella di scarica avviene in maniera del tutto automatica. La modalità avanzata invece, è stata pensata per un utente esperto che conosca molto bene l'intero ciclo di lavoro; infatti le singole fasi vengono avviate dall'utente quando necessario; così come l'irraggiamento dei singoli pellet vengono iniziati e terminati dall'utente.

2.1 Motori ed attuatori

La movimentazione dei pellet all'interno dell'esperimento IRIS è mediata attraverso 7 motori passo passo e 2 attuatori lineari (solenoidi); ciascun motore viene identificato con la sigla MX dove X=1,...,6 e ogni motore è associato ad un componente.

- **M1** → Charge Slider: movimenta lo scivolo che accoppia il sistema di caricamento e la corona circolare in cui vengono caricati i pellet.
- **M2** → Charge Buffer: movimenta un buffer che porta i pellet nel Charge Slider.
- **M3** → Discharge Slider: movimenta lo scivolo che accoppia la corona circolare in cui vengono caricati i pellet e il sistema di scarica.
- **M4** → Discharge Buffer: movimenta un buffer che porta i pellet dalla corona circolare al punto di detection.

- **M5** → Central Movement: corona circolare in cui vengono caricati i pellet.
- **M6** → Camera Alignment: sistema che accoppia il sistema dove è presente la corona circolare alla Implantatio Chamber.
- **M7** → Block Target: movimenta una linguetta metallica che blocca il pellet nella posizione di detection.

2.2 Charge - Irradiation - Discharge

In questa sezione si vuole spiegare più in dettaglio le fasi di caricamento, irraggiamento e scarica.

2.2.1 Charge

Il caricamento dei pellet coinvolge diversi motori tra cui il *Charge Slider* (M1), il *Charge Buffer* (M2) e il *Central Movement* (M5), i quali devono operare in modo coordinato al fine di garantire il corretto processo di caricamento.

La fase di caricamento può essere suddivisa in:

- Accoppiamento Charge slider (M1).
- Movimentazioni pellet da charge buffer (M2), al charge slider (M1) ed infine al central movement (M5).
- Disaccoppiamento Charge slider (M1).

La fase più complessa risulta essere la movimentazione dei pellet, in quanto richiede l'impiego combinato di *M1*, *M2* e *M5*. Per tale motivo viene adottato un *manager* che, in funzione dello stato operativo, gestisce le diverse richieste di movimentazione.

Di seguito viene illustrato il processo per il caricamento di un pellet.

1. Accoppiamento del *Charge Slider* (M1).
2. Rotazione del *Charge Buffer* (M2): un pellet viene caricato nel primo slot disponibile. Durante la rotazione il pellet viene trasferito nel *Charge Slider* per gravità. La rotazione si arresta al raggiungimento del finecorsa di M2.
3. Segnalazione al *Manager Motori* della disponibilità del pellet all'ingresso di M5.
4. L'*Attuatore 1* viene comandato in posizione di estrazione.
5. Il *Central Movement* (M5) viene fatto ruotare fino a una posizione tale da consentire l'ingresso del pellet nello slot.
6. M5 segnala l'avvenuto caricamento del pellet.
7. L'*Attuatore 1* viene comandato in posizione retratta.
8. M5 richiede al *Manager Motori* l'esecuzione dello *swing movement* di M1.
9. Il *Manager Motori* comanda l'esecuzione dello *swing movement*.
10. Al termine del movimento, il *Manager Motori* notifica a M5 l'avvenuto completamento e a M2 la possibilità di procedere con il caricamento di un nuovo pellet.
11. M2 ruota in senso opposto al fine di evitare la perdita di pellet durante la rotazione.

Il set di operazioni viene ripetuto per il secondo e il terzo pellet.

Lo *Swing Movement* è un movimento del *Charge slider* che viene eseguito ad ogni caricamento di un pellet. Nel caso in cui durante il caricamento del pellet in M5 non entri nello slot presente nel *Central Movement*, lo scivolo si disaccoppia per poi accoppiarsi nuovamente, in questo modo un eventuale pellet non propriamente caricato presente all'interfaccia di M1 e M5 viene "eiettato" e quindi non rappresenta essere un ostacolo problema per il caricamento del pellet successivo.

2.2.2 Irradiation

La fase denominata *irraggiamento* prevede, come prima operazione, l'accoppiamento della camera da vuoto mediante l'azionamento del motore *M6*. Una volta completato l'accoppiamento, è necessario generare il vuoto all'interno della camera. Tale operazione è gestita da un sistema esterno e non può essere effettuata direttamente tramite il software di controllo.

È di fondamentale importanza evitare la rottura improvvista del vuoto, poiché tale evento potrebbe causare gravi danni all'intera linea *SPES*. A tal fine, entra in gioco la comunicazione con il *Machine Protection System* (*MPS*): lo scambio di segnali di richiesta e di stato macchina con l'*MPS* consente di prevenire interruzioni non controllate del vuoto all'interno della camera. In particolare, una volta accoppiata la camera, il motore *M6*, unico elemento in grado di movimentarla, viene disalimentato per garantire la stabilità del sistema.

In prossimità dell'interfaccia con la linea *SPES* è presente un sistema costituito da una *Faraday Cup* e da un *collimatore*, montati sul medesimo asse. Ciò implica che, in ogni istante, soltanto uno dei due dispositivi possa trovarsi in linea con il fascio.

La *Faraday Cup* ha il compito di misurare la corrente del fascio, fornendo quindi una stima della sua intensità, mentre il collimatore è utilizzato per collimare il fascio sul pellet. Analogamente a quanto accade per la gestione del vuoto, anche il controllo del sistema *Faraday Cup + collimatore* è demandato a un sistema esterno e non è gestibile direttamente dal software di controllo.

L'irraggiamento delle singole pastiglie avviene in modalità sequenziale; il tempo di esposizione può essere definito a priori, con funzionamento automatico attraverso dei timer (modalità base), oppure determinato manualmente dall'operatore (modalità avanzata).

La sequenza delle operazioni previste sarà quindi:

1. Accoppiamento camera e posizionamento del primo pellet caricato in posizione di irraggiamento.
2. Richiesta vuoto e disabilitazione *M6*.
3. Misurazione corrente di fascio con *FC* (?).
4. Richiesta posizionamento collimatore.
5. Irraggiamento primo pellet.
6. Posizionamento secondo pellet.
7. Irraggiamento secondo pellet.
8. Posizionamento terzo pellet.
9. Irraggiamento terzo pellet.
10. Richiesta venting (togliere il vuoto) e richiesta abilitazione *M6*.
11. Disaccoppiamento camera e posizionamento della stessa in posizione per la scarica.

POSSIBILE MODIFICA SE CI E' DATO IL PERMESSO DI CONTROLLARE FC+COLL.

2.2.3 Discharge

La fase di *Discharge* rappresenta l'ultima fase del ciclo di lavoro e consiste nella movimentazione dei pellet irraggiati verso una postazione di misura offline dove sono presenti due differenti detector. Successivamente, i pellet vengono trasferiti in una provetta collocata all'estremità della guida in plastica lungo la quale scorrono.

L'ordine di scarica rispetta l'ordine di irraggiamento: la prima pastiglia irraggiata è quindi anche la prima a essere condotta nella posizione di misura. L'intervallo di tempo tra la conclusione dell'irraggiamento e l'avvio della misurazione viene monitorato e registrato, poiché costituisce un parametro importante per la tracciabilità dell'attività raccolta. La movimentazione dei pellet viene eseguita mediante l'utilizzo dei motori *M3*, *M4*, *M5* e *M7* e si divide in due parti:

- Scarica pellet dal Central movement (*M5*) al discharge slider (*M3*) ed infine nel discharge buffer (*M4*);

- Movimentazione dei pellet dal discharge buffer (M4) alla posizione di misurazione offline (M7);

Si ricorda che il *Discharge Buffer* è dotato di tre slot destinati ad accogliere i pellet. A causa di vincoli legati alla progettazione meccanica, è necessario che i pellet non vengano a trovarsi in prossimità l'uno dell'altro, al fine di evitare interferenze meccaniche e possibili sovrapposizioni. Per tale motivo è stato adottato un metodo di scarica da M5 che consente al *Central Movement* di trasferire ciascun pellet in uno degli slot del *Discharge Buffer*, garantendo così la presenza di un solo pellet per volta sulle guide di scorrimento.

Analogamente, nella seconda fase è previsto che soltanto un pellet sia presente nella posizione di misura offline. Tale accorgimento si rende necessario per evitare che l'attività registrata da un pellet interferisca con la misurazione dell'attività relativa a un altro campione.

Lista delle operazioni svolte per la prima parte:

1. Il *Central Movement* viene portato in una posizione nota ovvero nella posizione di zero.
2. L'*Attuatore 2* viene comandato in posizione estratta e il *Central Movement* ruota fino alla prima posizione di scarica.
3. Il pellet cade nel *Discharge Slider* e raggiunge l'interfaccia con il *Discharge Buffer*.
4. Il *Discharge Slider* esegue lo *swing movement*.
5. Il *Discharge Buffer*, dalla posizione di *home*, ruota fino alla prima posizione di scarica.
6. Il pellet entra nel primo slot.
7. L'*Attuatore 2* viene comandato in posizione retratta.
8. Il *Discharge Buffer* ruota fino a una posizione intermedia tra il primo e il secondo slot.
9. Il *Central Movement* ruota fino alla seconda posizione di scarica e l'*Attuatore 2* viene comandato in posizione estratta.
10. Il pellet cade nel *Discharge Slider* e raggiunge l'interfaccia con il *Discharge Buffer*.
11. Il *Discharge Slider* esegue lo *swing movement*.
12. Il *Discharge Buffer* ruota fino a una posizione intermedia tra il secondo e il terzo slot.
13. Il pellet entra nel secondo slot.
14. L'*Attuatore 2* viene comandato in posizione retratta.
15. Il *Central Movement* ruota fino alla terza posizione di scarica e l'*Attuatore 2* viene comandato in posizione estratta.
16. Il pellet cade nel *Discharge Slider* e raggiunge l'interfaccia con il *Discharge Buffer*.
17. Il *Discharge Slider* esegue lo *swing movement*.

Operazioni svolte per la seconda parte:

1. Viene bloccata la posizione di misurazione offline (*M7*).
2. Il *Discharge Buffer* ruota fino a scaricare il primo pellet e si arresta.
3. Avvio della misurazione.
4. Conclusione della misurazione.

Le stesse operazioni vengono eseguite per il secondo e il terzo pellet.

3 Lasal Class 2 - Control programming

LASAL Class 2 è l'ambiente di sviluppo utilizzato per la progettazione e la programmazione del sistema di controllo per la movimentazione dei pellet all'interno dell'esperimento IRIS. La piattaforma integra la programmazione orientata agli oggetti, conforme allo standard IEC 61131-3, con una rappresentazione grafica dei componenti realmente presenti. I singoli elementi fisici della macchina vengono modellati come oggetti software, chiamati anche *classi*, consentendo una corrispondenza diretta tra hardware e logica di controllo. Questa struttura facilita l'implementazione di logiche complesse, migliorando la modularità, la leggibilità del codice e la riutilizzabilità dei componenti software.

3.1 Motors and Actuators

La movimentazione dei pellet all'interno dell'esperimento *IRIS* è affidata a un insieme di **sette motori passo-passo** e **due attuatori elettromeccanici** (solenoidi). Nonostante le differenze fisiche e funzionali tra i dispositivi, per ciascun motore è stata sviluppata una classe dedicata che ne descrive in modo specifico il comportamento. Questo approccio è stato necessario poiché ogni motore è associato a una funzione distinta all'interno del sistema.

Nella progettazione delle classi software sono stati considerati diversi fattori, tra cui:

- Relazione tra il **motore** e il **componente hardware** da movimentare;
- Presenza di uno o più **finecorsa** (limit switch, LS);
- Relazione tra il **tipo di movimento richiesto** e la **fase del processo** in cui il movimento è richiesto.

Per garantire maggiore uniformità è stato deciso di adottare una **struttura FSM principale omogenea** per tutti i motori. Tale struttura è composta dai seguenti stati fondamentali:

- **Idle** → Stato di attesa comandi.
- **Homing** → Stato di inizializzazione e posizionamento del motore nella posizione di riferimento.
- **Movement** → Stato dell'esecuzione del movimento richiesto in base alla sequenza attiva.
- **Error** → Stato di errore durante il funzionamento.
- **Reset** → Stato in cui si attende il comando di reset per consentire la ripresa della normale funzionalità dopo un'interruzione.
- **stop** → Stato in cui vengono inviati i comandi di stop e spegnimento motore.
- **stopped** → Stato in cui il motore è fermo e attende una richiesta di reset.

Questa organizzazione consente una gestione modulare e standardizzata della movimentazione, facilitando sia l'integrazione delle singole classi nel sistema complessivo, sia la manutenzione e l'estensione futura del software.

Andiamo a vedere più in dettaglio le caratteristiche principali dei diversi motori.

Charge & Discharge slider

I motori associati al *Charge slider* - *M1* e al *Discharge slider* - *M3* eseguono le seguenti tipologie di movimento:

- **Homing**: movimento verso il finecorsa di riferimento (limit switch, LS) per la calibrazione iniziale.
- **Movement**:
 - *Accoppiamento* → Spostamento verso il LS di accoppiamento.
 - *Disaccoppiamento* → Spostamento verso il LS di disaccoppiamento.
 - *Swing movement* → Doppio movimento relativo alla posizione di accoppiamento per evitare eventuali blocchi del target durante il percorso.

I movimenti richiesti sono quindi movimentazioni tra un LS e l'altro e movimentazioni relative rispetto a una posizione di riferimento.

Chamber Alignment

Per il motore associato all'*allineamento camera - M6* i movimenti previsti sono:

- **Homing**
- **Movement:**

- *Accoppiamento* → Spostamento verso il LS di accoppiamento.
- *Disaccoppiamento* Spostamento verso il LS di disaccoppiamento

Block Target

Per il motore associato al *block target*, i movimenti sono:

- **Movement:**
- *Accoppiamento* → Posizionamento in configurazione di blocco.
- *Disaccoppiamento* → Posizionamento in configurazione di sblocco.

Central Movement

Il motore associato al *central movement* esegue i seguenti movimenti:

- **Homing**
- **Movement:**
 - *Carica* → Movimentazioni per il caricamento dei target nella camera.
 - *Irraggiamento* → Movimentazioni per portare sequenzialmente i target nella posizione di irraggiamento.
 - *Scarica* → Movimentazioni per scaricare i target dal *Central Movement*.

La caratterizzazione di questo motore presenta un'elevata complessità, in quanto i suoi movimenti non sono autonomi ma vincolati alle dinamiche degli altri motori ed attuatori del sistema. Di conseguenza, è stata sviluppata una logica di controllo dedicata che consente l'interazione con il **Manager Motori**, attraverso il quale il motore comunica il proprio stato operativo e coordina le richieste di movimentazione con gli altri motori coinvolti, garantendo il corretto sincronismo e la coerenza funzionale del sistema.

Charge & Discharge Buffer

Per i motori associati ai buffer di carica (*charge buffer - M2*) e di scarica (*discharge buffer - M4*) sono previste le seguenti modalità operative:

- **Homing**
- **Movement:**
 - Movimenti per la fase di carica (*charge buffer*);
 - Movimenti per la fase di scarica (*discharge buffer*).

Analogamente al motore associato al *Central Movement*, anche questi attuatori devono operare in maniera coordinata con gli altri elementi del sistema. Ciò è reso possibile mediante funzioni apposite che interagiscono con la classe *manager Motori*, responsabile della gestione centralizzata delle richieste di movimento e dei relativi feedback.

Attuatori

Gli attuatori, per loro natura, possono assumere esclusivamente due stati discreti di posizione:

- **Estratto**, corrispondente alla posizione di massima estensione;
- **Retratto**, corrispondente alla posizione di riposo o minima estensione.

3.2 Struttura software di controllo

Per la stesura del codice di controllo è stato adottato un approccio modulare e gerarchico, basato sulla scomposizione del problema principale in sottoproblemi più semplici da implementare e testare. All'interno dell'ambiente di sviluppo LASAL Class 2, questo approccio si traduce nella creazione di *reti logiche*, ognuna delle quali è composta da classi che rappresentano funzionalità o componenti specifici. Le classi all'interno di una rete possono interagire tra loro e con classi appartenenti ad altre reti, consentendo così un'elevata flessibilità, riusabilità e una chiara strutturazione del progetto software.

Sono state implementate le seguenti reti:

- Hardware Network
- Automatic Network
- Manual Network
- Axis Profiles Network
- Modbuss Communication Network
- Settings Network
- Timers Network

3.3 Hardware network

In figura 3 è mostrata la rete denominata *Hardware Network*. Questa rete viene generata automaticamente da **LASAL** nel momento in cui i moduli **Sigmatek**, fisicamente connessi tra loro, vengono collegati al PC dove il software viene sviluppato. Una volta riconosciuti i moduli, questi vengono letti e importati all'interno del progetto.

L'integrazione dei moduli nella rete hardware consente all'utente di utilizzare **funzionalità predefinite** messe a disposizione dall'ambiente di sviluppo, come ad esempio, i comandi di movimentazione per i motori (movimenti assoluti, relativi, referenziati, ecc.). Inoltre, è possibile associare **classi definite dall'utente (user defined)** ai componenti hardware, permettendo un collegamento diretto tra il software e i dispositivi fisici sul campo, come ad esempio ingressi e uscite digitali (*Digital I/O*).

I moduli utilizzati sono:

- 1 Modulo CPU CP-112
- 1 Modulo Digital Input DI-160
- 1 Modulo Digital Output TO-127
- 1 Modulo Analog Input
- 9 Moduli Stepper Motor ST-151

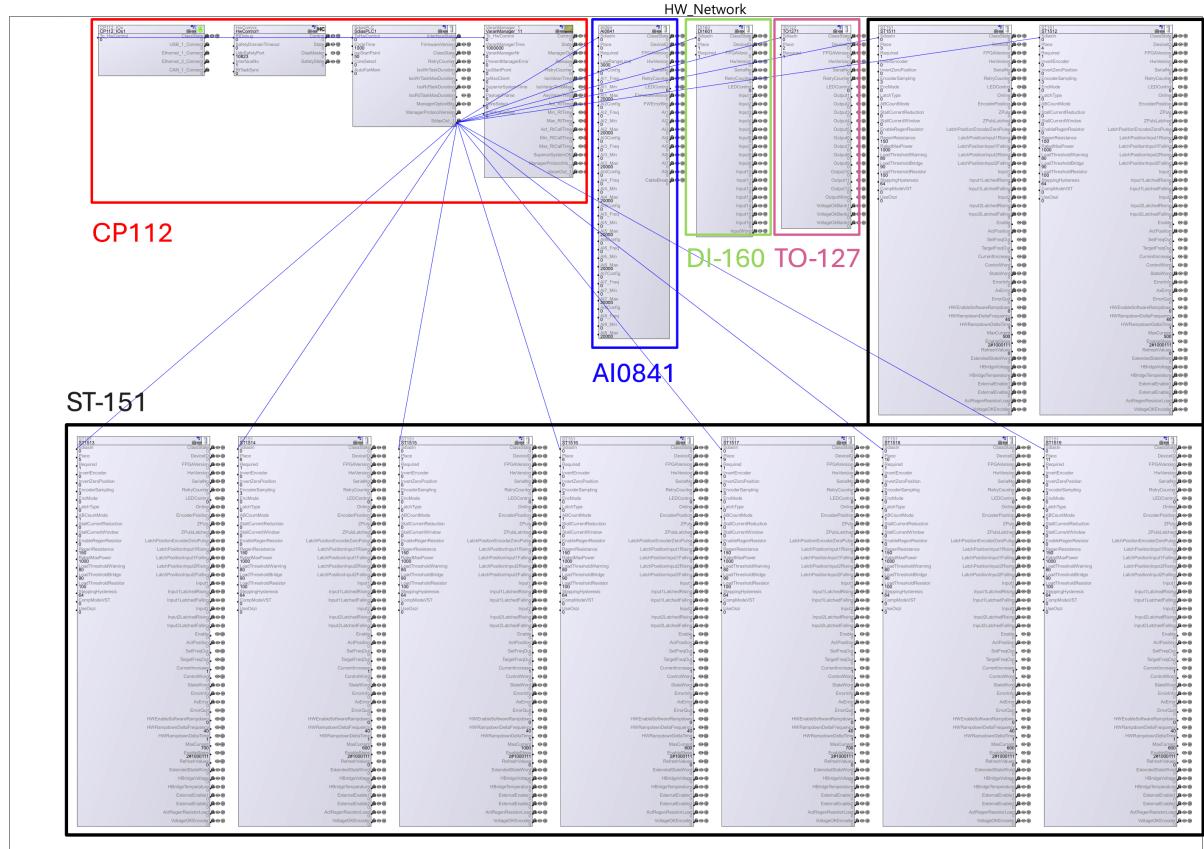


Figure 3: Hardware Network

3.4 Automatic Network

La rete denominata *Automatic Network* rappresenta la parte centrale e più articolata del progetto, occupandosi della gestione del processo di movimentazione in modalità automatica. L'obiettivo principale di questa rete è ridurre al minimo l'intervento dell'operatore, limitandolo all'interazione con l'interfaccia grafica (GUI), attraverso la quale può inviare comandi predefiniti. Il software interpreta tali comandi e li elabora eseguendo automaticamente le azioni richieste.

Per facilitare l'implementazione, è stata adottata una struttura modulare, che ha permesso di suddividere il problema in componenti funzionali distinti. Sono state quindi create diverse classi, ognuna delle quali ha responsabilità specifiche all'interno del sistema.

Come illustrato in figura 4, la rete è composta dalle seguenti classi principali:

- **Global Manager**
- **Main Sequence**
- **Motors' Manager**
- **Motors and Actuators**

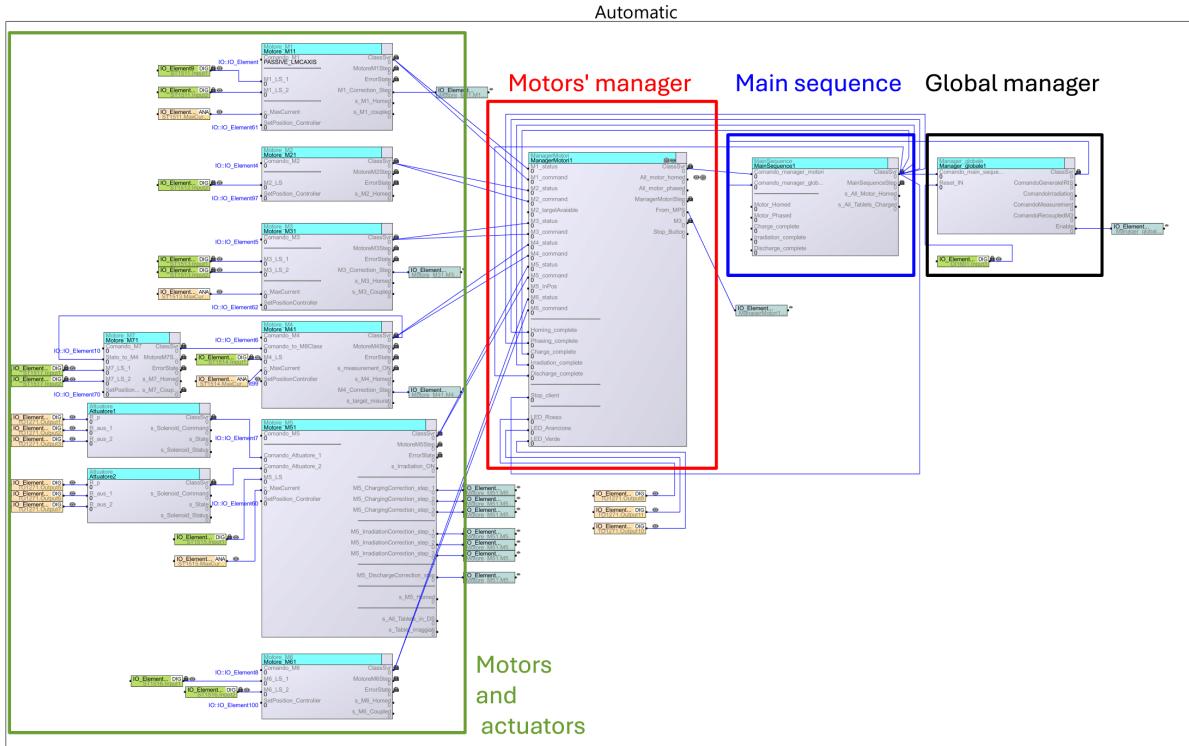


Figure 4: Automatic Network

3.4.1 Global Manager

Questa classe funge da **interfaccia logica tra l'utente e il software di controllo**. Al suo interno è stata implementata una *Finite State Machine* (FSM) che gestisce la transizione tra diversi stati operativi, ciascuno dei quali rappresenta una fase specifica del ciclo di lavoro. Gli stati principali previsti dalla FSM sono:

- **Idle**
 - **Homing**
 - **Charge**
 - **Irradiation**
 - Irradiation Idle
 - Irradiation Start
 - Irradiation Over
 - **Discharge**
 - Measurement Idle
 - Measurement Start
 - Measurement Over
 - **Stop**
 - **Error State**
 - **Reset**

Alcuni stati, come *Irradiation* e *Discharge*, contengono a loro volta una FSM interna per gestire sottostati e comandi utente più dettagliati, garantendo un controllo fine delle operazioni complesse.

Il **Global Manager** riceve i comandi dall'utente (tramite GUI o altri input) e, in base allo stato corrente e al comando ricevuto, invia alla classe *Main Sequence* le istruzioni necessarie per determinare lo stato operativo successivo. Questo consente una gestione sequenziale e controllata del processo automatico.

Le funzioni implementate sono:

- SetSequenceStep
- SetIrradiationState

Entrambe le funzioni vengono utilizzate dalla classe *Main sequence* per riportare lo stato macchina nello stato di idle o nello stato successivo rispetto a quello attuale a seconda del tipo di controllo selezionato dall'utente.

3.4.2 Main Sequence

Questa classe funge da interfaccia tra la classe *Global Manager* e il **Manager Motori**. La funzione principale di questa classe è quella di ricevere, dal *Global Manager*, il comando utente già elaborato e determinare, sulla base di questo, lo stato corretto della FSM principale in cui transitare.

La *Main Sequence* rappresenta quindi un modulo intermedio tra i comandi provenienti dal *Global Manager* e l'effettiva attivazione della logica di controllo motori, gestita dalla classe *Motors' Manager*.

Analogamente al *Global Manager* è stata implementata una **Finite State Machine** (FSM) principale, accompagnata da FSM secondarie, specifiche per determinati stati.

Gli stati previsti dalla FSM principale sono:

- **Idle**
 - Cmd_idle
 - Cmd_start
 - Cmd_charge
 - Cmd_irradiation
 - Cmd_discharge
 - Cmd_error
 - Cmd_stop
 - Cmd_reset

- **Homing**

- **Charge**

- **Irradiation**

- **Discharge**

- **Stop**

- **Error**

- **Reset**

La FSM è progettata in modo che, all'avvio del sistema, lo stato iniziale sia quello di **Idle**, che rappresenta la condizione di attesa comandi. A partire da questo stato, la macchina può transitare verso altri stati in funzione dei comandi ricevuti dal *Global Manager*.

Ogni stato operativo è monitorato tramite un sistema di **feedback** che consente di rilevare l'avvenuto completamento dell'attività in corso. Al termine di ciascuna sequenza, viene inviata una notifica al *Global Manager*, che aggiorna lo stato generale del sistema e permette alla FSM di tornare in attesa di nuovi comandi, riportandosi nello stato **Idle**.

Per maggior chiarezza in figura 5 viene riportato parte del codice che descrive la classe *MainSequence*.

```

MainSequence

case MainSequenceStep of
  Idle:
    if Comando_manager_motori.GetErrorState() & vFlag=0 then
      vFlag:=1;
      ActCommand:=Cmd_error;
      Comando_manager_globale.SetSequenceStep(Cmd:=Manager_globale::ErrorState);
    end_if;

  case ActCommand of
    Cmd_Idle:
    Cmd_Start:
      if v_motor_homed=0 then
        MainSequenceStep:=Homing;
      end_if;

    Cmd_Charge:
      MainSequenceStep:=Charge;
    Cmd_irradiation:
      MainSequenceStep:=Irradiation;
    Cmd_Discharge:
      MainSequenceStep:=Discharge;
    Cmd_Error:
    Cmd_Stop:
      MainSequenceStep:=Stop;
    Cmd_Reset:
      MainSequenceStep:=Reset;
  end_case;
  =====
  Homing:
    if Comando_manager_motori.GetErrorState() then
      MainSequenceStep:=Idle;
      ActCommand:=Cmd_error;
      Comando_manager_globale.SetSequenceStep(Cmd:=Manager_globale::Idle);
    end_if;

    Comando_manager_motori.SetSequenceState(Cmd_ManagerMotori:=Cmd_ManagerMotori_homing, StartCase:=10);
    if v_Motor_Homed then
      Comando_manager_globale.SetSequenceStep(Cmd:=Manager_globale::Idle);
      ActCommand:=Cmd_Idle;
      MainSequenceStep:=Idle;
    end_if;

```

Secondary FSM

Main FSM

Figure 5: Main Sequence

All'interno di questa classe viene gestita la distinzione tra la modalità operativa *base* e quella *avanzata*. A livello software, tale distinzione si traduce in un diverso comportamento della macchina a stati finiti (FSM) al termine di ciascuna fase: nella modalità base, infatti, la FSM viene direttamente instradata verso lo stato successivo, evitando il ritorno allo stato di *idle* e quindi alla condizione di attesa di comandi esterni. Questo comportamento è illustrato in figura 6, dove è possibile osservare che, se la modalità base è abilitata, al termine della fase di carica il sistema transita direttamente alla fase di irraggiamento, senza passare dallo stato di *idle*.

```

MainSequence

Charge:
if Comando_manager_motori.GetErrorState() then
    MainSequenceStep:=Idle;
    ActCommand:=Cmd_error;
    Comando_manager_globale.SetSequenceStep(Cmd:=Manager_globale::Idle);
end_if;

IF v_charge_complete=2 THEN
    if v_EasVGUT then
        Comando_manager_globale.SetSequenceStep(Cmd:=Manager_globale::Irradiation);
    else
        Comando_manager_globale.SetSequenceStep(Cmd:=Manager_globale::Idle);
    end_if;
    MainSequenceStep:=Idle;
    ActCommand:=Cmd_idle;
ELSE
    Comando_manager_motori.SetSequenceState(Cmd_ManagerMotori:=ManagerMotori::Cmd_ManagerMotori_charge,
StartCase:=10);
END_IF;

```

Figure 6: Distinzione modalità operativa base e avanzata

Le funzioni implementate in questa classe sono le seguenti:

1. SetHomingStatus
2. SetChargeStatus
3. SetIrradiationStatus
4. SetDischargeStatus
5. SetIrradiationStartOver
6. SetMeasurementStartOver
7. SetStop
8. SetSequenceState
9. SetSequenceState

Le funzioni dalla 1 alla 4 vengono utilizzate dalla classe **Manager Motori** per notificare il completamento delle rispettive fasi operative. Le funzioni 5 e 6, invece, trasmettono le informazioni relative all'inizio e alla fine delle fasi intermedie nell'irraggiamento e nella misura, fornite dal **Manager globale**. Infine, le funzioni 8 e 9 sono impiegate dalla classe **Manager Motori** per impostare la classe **MainSequence** nello stato di arresto.

3.4.3 Motors' manager

La classe *Motors' Manager* ha il compito di **coordinare l'attivazione e il controllo dei motori e degli attuatori** presenti sul campo. In base al processo selezionato dall'utente, questa classe stabilisce sia l'*ordine di attivazione* dei dispositivi sia il *tipo di movimento* da eseguire. Il tutto avviene sfruttando un sistema di **feedback** fornito dalle classi associate ai singoli motori e attuatori, permettendo così una gestione dinamica e affidabile delle operazioni.

Analogamente ad altre parti del progetto, anche in questa classe è stata implementata una **FSM principale**, che definisce i macro-stati del processo (es. *Idle*, *Homing*, ecc.). All'interno di ciascun stato principale, è presente una FSM secondaria che suddivide in maniera sequenziale i singoli passi da eseguire per completare correttamente il processo.

In figura 7 è riportata una porzione della FSM principale, in cui sono visibili gli stati *Idle* e *Homing*. All'interno dello stato *Homing*, ad esempio, è presente una FSM secondaria composta da **nove step**. Ogni step rappresenta la fase di attivazione di un singolo motore per portarlo in posizione di homing. Il passaggio da uno step al successivo è condizionato dal corretto completamento della fase precedente, verificato tramite feedback. Al termine della sequenza, quando tutti i motori risultano essere nella posizione di homing, il sistema ritorna automaticamente nello stato *Idle*, pronto a ricevere nuovi comandi.



Figure 7: Motors' manager

Le funzioni implementate sono le seguenti:

1. SetSequenceState
 2. SetIrradiationStartOver
 3. SetMeasurementStartOver
 4. GetErrorState

La funzione 1 viene utilizzata dalla classe `MainSequence` per comandare lo stato in cui la classe `ManagerMotori` deve entrare. Le funzioni 2 e 3 hanno il compito analogo alle omonime presenti nella classe `MainSequence`; in questo caso le informazioni vengono passate da `MainSequence` alla classe `ManagerMotori`. La funzione 4 viene anch'essa utilizzata dalla classe `MainSequence` ma per rilevare se la classe `ManagerMotori` sia entrata o meno nello stato di errore.

Più interessante può essere il meccanismo attraverso il quale vengono "attivati" i diversi motori a seconda dello stato macchina in cui ci si trova.

Homing

La procedura di homing viene eseguita su un motore alla volta, al fine di evitare possibili interferenze durante la movimentazione. In questa fase, inoltre, viene comunicato a determinati motori se la modalità di caricamento selezionata dall'operatore prevede 1 oppure 3 pellet, figura 8.

```

ManagerMotori

Cmd_ManagerMotori_homing:

    //dò ad ogni motore il comando di homing; per sicurezza facciamo fare questa operazione ad un motore alla volta.
    if M1_status.GetErrorState()=1 | M2_status.GetErrorState()=1 | M3_status.GetErrorState()=1 |
    M4_status.GetErrorState()=1 | M5_status.GetErrorState()=1 | M6_status.GetErrorState()=1 | Stop_Button.Read()=1 then
        ManagerMotoriStep:=Cmd_ManagerMotori_stop;
        v_Errorstate:=1;
        Homing_motor_Mx:=0;
    end_if;

    case Homing_motor_Mx of
        0:
            Homing_motor_Mx:=v_StartHomingPhase;
            v_allMotorHomed:=1;
            Homing_complete.SetHomingStatus(Status:=1); //homing in esecuzione
            c MPS_Req.write(input:=1); //Richiesta a MPS per muovere M6

    10://Eseguo procedura di homing per CHARGE SLIDER
        M1_command.SetSequenceState(Cmd_motore:=Cmd_motore_M1_homing, Motion_type:=0, StartCase:=10);
        //se motore homed allora passo al motore successivo
        if M1_status.GetHomedState() then
            M1_status:=M1_status.GetHomedState(); Controllo se M1 ha completato l'homing
            Homing_motor_Mx:=20;
        end_if;

    20://Eseguo procedura di homing per CHARGE BUFFER
        M2_command.SetSequenceState(Cmd_motore:=Motore_M2::Cmd_motore_M2_homing, StartCase:=10);
        c_1_3_pelletMode_M2.write(input:=v_1_3_pelletMode); //1/3 pellet mode per M2
        if M2_status.GetHomedState() then
            M2_status:=M2_status.GetHomedState();
            Homing_motor_Mx:=30;
        end_if;

    30://Eseguo procedura di homing per DISCHARGE SLIDER
        M3_command.SetSequenceState(Cmd_motore:=Motore_M3::Cmd_motore_M3_homing, Motion_Type:=0, StartCase:=10);
        if M3_status.GetHomedState() then
            M3_status:=M3_status.GetHomedState();
            Homing_motor_Mx:=40;
        end_if;
    // 40://Eseguo procedura di homing per DISCHARGE BUFFER
        M4_command.SetSequenceState(Cmd_motore:=Motore_M4::Cmd_motore_M4_homing, Motion_Type:=0, StartCase:=10);
        c_1_3_pelletMode_M4.write(input:=v_1_3_pelletMode); //1/3 pellet mode per M4
        if M4_status.GetHomedState() then
            M4_status:=M4_status.GetHomedState();
            Homing_motor_Mx:=50;
        end_if;
    
```

Figure 8: Homing Manager Motori

Charge

Durante la fase di *Charge*, figura 9, il processo ha inizio con l'accoppiamento del *Charge slider - M1* nello stato 10, seguito dall'attivazione dei motori M2 e M5 nello stato 20. Nello stato 30, il sistema permane all'interno di una struttura *if* finché il motore M5 non segnala l'avvenuto caricamento dei pellet. In questa fase, viene gestato uno scambio di informazioni tra i motori M2 e M5 al fine di coordinare i movimenti necessari; inoltre, viene gestita la richiesta, da parte di M2, dell'esecuzione dello *swing movement - M1*. Al termine del caricamento, nello stato 40, viene avviata la procedura di disaccoppiamento del *Charge slider - M1*. Infine, nello stato 50, viene notificato alla classe *MainSequence* il completamento della fase di carica.

```

ManagerMotori

//=====
Cmd_ManagerMotori_charge:
    if M1_status.GetErrorState()=1 | M2_status.GetErrorState()=1 | M3_status.GetErrorState()=1 |
    M4_status.GetErrorState()=1 | M5_status.GetErrorState()=1 | M6_status.GetErrorState()=1 | Stop_Button.Read()=1 then
        ManagerMotoriStep:=Cmd_ManagerMotori_stop;
        v_ErrorState:=1;
        Charge_step:=0;
    end_if;

    case Charge_step of
        0: //idle
            Charge_step:=v_StartChargePhase;
            Charge_complete.SetChargeStatus(Status:=1);
            LED_Rosso.Write(input:=1);

        10: //AVVIO M1 - ACCOPPIO
            if M1_status.GetCoupledState()=0 then
                M1_command.SetSequenceState(Cmd_motore:=Motore_M1::Cmd_motore_M1_movement, Motion_type:=0, StartCase:=10);
            //Motion_type:=0 disacc->acc || Motion_type:=1 acc->disacc
            elseif M1_status.GetCoupledState()=2 then
                Charge_step:=20;
            end_if;

        20://M1 E' IN POSIZIONE
            //Comando ai motori M2 e M5 ti entrare nello stato di movimentazione
            M2_command.SetSequenceState(Cmd_motore:=Motore_M2::Cmd_motore_M2_movement, StartCase:=10);
            M5_command.SetSequenceState(Cmd_motore:=Motore_M5::Cmd_motore_M5_movement_CHARGING, Motion_Type:=0,
            Charge_step:=30);

        30:
            //se non è finita la fase di carica allora è necessario continuare ad aggiornare lo stato di M5 e passarlo a M2 -
            //CICLICAMENTE AGGIORNO LO STATO DI M5 DA PASSARE A M2
            if M5_command.GetStateCharging()=FALSE then
                v_M5_InPos:=M5_commandGetPositionChargeState(); //assegno ad una variabile lo stato di m5
                //M5_InPos.Write(input:=v_M5_InPos); //scrivo sul client lo stato di m5
                M2_command.ReadM5PositionState(Param:=M5_InPos); //passo a m2 lo stato di m5

                v_target_in_ChargeSliderM2:=M2_command.GetTargetInSlider();
                //M2_targetAvailable.write(input:=v_target_in_ChargeSliderM2);
                M5_command.ReadTargetM2(Status:=M2_targetAvailable);

                if M2_command.Req_M1_Mov() then
                    M1_command.SetSequenceState(Cmd_motore:=Motore_M1::Cmd_motore_M1_movement, Motion_type:=2, StartCase:=10);
                    if M1_command.GetMovDone() then
                        M2_command.ACK_M1_Mov(Param:=1);
                        M5_command.ACK_M1_Mov(Param:=1);
                        Controllo la richiesta dello swing movement (M1)
                        da parte di M2
                    end_if;
                end_if;

                else
                    //3 target sono stati caricati
                    Charge_step:=40;
                end_if;
            end_if;

        40: //DISACCOPPIAMENTO M1
            M1_command.SetSequenceState(Cmd_motore:=Motore_M1::Cmd_motore_M1_movement, Motion_type:=1, StartCase:=10);
            if M1_status.GetCoupledState()=0 then
                Charge_step:=50;
            end_if;

        50:
            Charge_complete.SetChargeStatus(Status:=2);
            ManagerMotoriStep:=Cmd_ManagerMotori_idle;
            Charge_step:=0;
            LED_Rosso.Write(input:=0);
            v_StartChargePhase:=0;

    end_case;

```

Figure 9: Charge Manager Motori

Irradiation

La fase di irraggiamento, illustrata in figura 10, ha inizio con l'accoppiamento del sistema piatto/*Implantation chamber* tramite il motore M6, nello stato 20. Successivamente, nello stato 30, viene avviato il motore M5 e, qualora M6 abbia completato correttamente l'accoppiamento, viene disattivata la richiesta al *Machine Protection System - MPS* di movimentazione per quest'ultimo.

Nel successivo stato 35, il motore M5 riceve aggiornamenti costanti del comando di avvio/arresto dell'irraggiamento, trasmesso dalla classe *MainSequence*. Quando tutti i pellet sono stati tutti correttamente irraggiati, è necessaria una richiesta di autorizzazione al *MPS* per disaccoppiare la camera.

Una volta ricevuto un acknowledgment positivo da parte dell'*MPS*, nello stato 40 viene avviata la procedura di disaccoppiamento del motore M6. Al termine di questa operazione, nello stato 50, viene notificato alla classe *MainSequence* il completamento della fase di irraggiamento.

```
ManagerMotori

Cmd_ManagerMotori_irradiation:
    if M1_status.GetErrorState()=1 | M2_status.GetErrorState()=1 | M3_status.GetErrorState()=1 | 
    M4_status.GetErrorState()=1 | M5_status.GetErrorState()=1 | M6_status.GetErrorState()=1 | Stop_Button.Read()=1 then
        ManagerMotoriStep:=Cmd_ManagerMotori_stop;
        v_Errorstate:=1;
        Irradiation_Step:=0;
    end_if;

    case Irradiation_Step of
        0://idle
            Irradiation_Step:=v_StartIrradiationPhase;
            LED_Arancione.Write(input:=1);

        10:
            Irradiation_Step:=20;

        20://ACCOPPIAMENTO ATTRaverso M6
            M6_command.SetSequenceState(Cmd_motore:=Motore_M6::Cmd_motore_M6_movement, Motion_type:=0,
            StartCase:=10); //AVVIO L'ACCOPPIAMENTO
            if M6_status.GetCoupledState() then
                Irradiation_Step:=30;
            end_if;

        30://AVVIO DEL POSIZIONAMENTO DEI VARI TARGET
            M5_command.SetSequenceState(Cmd_motore:=Motore_M5::Cmd_motore_M5_movement_IRRADIATION, Motion_type:=0,
            StartCase:=5);
            if M6_status.GetCoupledState()=2 then
                Irradiation_Step:=35;
                C_MPS_Req.Write(input:=0); //quando camera accoppiata abbassiamo la Req -> anche Ack 0 e quindi motore
                disabilitato;
            end_if;
        35:
            M5_command.GetIrradiationStartOver(Status:=v_IrradiationStartOver);

            if M5_status.GetIrradiationStatus() then
                C_MPS_Req.Write(input:=1); //RICHIESTA AUTOMATICA A MPS
                Irradiation_Step:=40;
            end_if;

        40:
            if From_MPS.Read()=1 then
                M6_command.SetSequenceState(Cmd_motore:=Motore_M6::Cmd_motore_M6_movement, Motion_type:=1,
                StartCase:=10); //AVVIO DISACCOPPIAMENTO
                Irradiation_Step:=50;
            end_if;

        50://se ho completato il disaccoppiamento allora posso terminare la fase di irraggiamento
            if M6_status.GetCoupledState()=false then
                Irradiation_complete.SetIrradiationStatus(Status:=1); //dico al main sequence che la fase di irraggiamento è
                stata completata
                Irradiation_Step:=0;
                v_StartIrradiationPhase:=0;
                ManagerMotoriStep:=Cmd_ManagerMotori_idle;
                LED_Arancione.Write(input:=0);
            end_if;
    end_case;
```

Figure 10: Irradiation Manager Motori

Discharge

La fase di scaricamento, illustrata in figura 11, ha inizio con l'accoppiamento del *Discharge slider - M3* nello stato 10, seguito dall'avvio del motore M5 nello stato 20. Nello stato 30 viene impiegata una struttura logica simile a quella adottata durante la fase di *Charge*: una struttura *if* consente alla macchina a stati di rimanere in questo stato finché tutti i pellet non sono stati completamente scaricati.

Durante questa fase, il motore M5 richiede le movimentazione sia di M3 sia del *Discharge buffer - M4*, in modo da trasferire le pastiglie dalla corona circolare al punto di rilevamento. Successivamente, nello stato 35, il motore M4 riceve aggiornamenti costanti del comando di *Start/Stop Measurement* proveniente dalla classe *MainSequence*. Una volta che tutti i pellet hanno attraversato il punto di rilevamento, viene avviata la procedura di disaccoppiamento di M3, corrispondente allo stato 40. Infine, al termine di questa operazione, viene notificato alla classe *MainSequence* il completamento della fase di scaricamento.

```

ManagerMotori

Cmd_ManagerMotori_Discharge:
    if M1_Status.GetErrorState()=1 | M2_Status.GetErrorState()=1 | M3_Status.GetErrorState()=1 |
    M4_Status.GetErrorState()=1 | M5_Status.GetErrorState()=1 | M6_Status.GetErrorState()=1 | Stop_Button.Read()=1 then
        ManagerMotoriStep:=Cmd_ManagerMotori_Stop;
        v_ErrorState:=1;
        Discharge_step:=0;
    end_if;

    case Discharge_step of

        0://IDLE
            Discharge_step:=v_StartDischargePhase;
            LED_Verde.write(input:=1);

        10://Accoppio M3
            if M3_Status.GetCoupledState()=0 then //se disaccoppiato
                M3_Command.SetSequenceState(Cmd_Motore:=Motore_M3::Cmd_motore_M3_movement, Motion_Type:=0,
                StartCase:=10); //comando di accoppiamento
            elsif M3_Status.GetCoupledState()=2 then
                Discharge_step:=20;
            end_if;

        20://prima parte di scarica
            if M3_Command.GetCoupledState()=2 then
                M5_Command.SetSequenceState(Cmd_Motore:=Motore_M5::Cmd_motore_M5_movement_DISCHARGING, Motion_type:=0,
                StartCase:=10);
            else_if Discharge_step:=30;
            end_if;

        30:
            if M5_Status.GetStateDischarging()=2 then //SE FINITO DI SCARICARE TUTTO ALLORA
                M4_Command.SetSequenceState(Cmd_Motore:=Motore_M4::Cmd_motore_M4_movement, Motion_type:=0,
                Discharge_step:=35;
            else
                if M5_Command.Req_M4_Mov() then //per richiedere movimentazione del discharge buffer
                    M4_Command.SetSequenceState(Cmd_Motore:=Motore_M4::Cmd_motore_M4_movement_2, Motion_Type:=1,
                    if M4_Command.GetMoveDone() then
                        M5_Command.ACK_M4_Mov(Param:=1);
                    end_if;
                end_if;

                if M5_Command.Req_M3_Mov() then //per richiedere swing movement del discharge slider
                    M3_Command.SetSequenceState(Cmd_Motore:=Motore_M3::Cmd_motore_M3_movement, Motion_type:=2,
                    if M3_Command.GetMoveDone() then
                        M5_Command.ACK_M3_Mov(Param:=1);
                    end_if;
                end_if;
            end_if;

        35:
            M4_Command.GetMisurationStartOver(Status:=v_MisurationStartOver);

        if M4_Status.SetDischargeOver() then
            Discharge_step:=40;
        end_if;

        40:
            if M3_Status.GetCoupledState()=2 then //se accoppiato
                M3_Command.SetSequenceState(Cmd_Motore:=Motore_M3::Cmd_motore_M3_movement, Motion_Type:=1,
                StartCase:=10); //comando di disaccoppiamento
            elsif M3_Status.GetCoupledState()=FALSE then
                Discharge_step:=45;
            end_if;

        45:
            if M3_Status.GetCoupledState()=FALSE then
                Discharge_step:=50;
            end_if;

        50:
            Discharge_complete.SetDischargeStatus(Status:=1);
            Discharge_step:=0;
            v_StartDischargePhase:=0;
            ManagerMotoriStep:=Cmd_ManagerMotori_Idle;
            LED_Verde.write(input:=0);
    end_case;

```

Figure 11: Discahrge Manager Motori

3.5 Manual network

La parte relativa alla movimentazione manuale risulta di più semplice implementazione, in quanto non richiede un'interazione sincrona tra i diversi motori. Per questo motivo, la struttura delle classi associate ai motori è sostanzialmente uniforme; eventuali differenze tra alcune di esse sono dovute a particolari componenti hardware presenti sul campo, come i finecorsa (LS, *Limit Switch*), che rendono necessarie alcune modifiche al comportamento dei motori per adattarsi alla configurazione specifica.

Sono quindi presenti due tipologie di classi:

- Motori con due finecorsa;
- Motori con un finecorsa.

Le funzionalità implementate sono le seguenti:

- Power On
- Power Off
- Selezione della posizione desiderata
- Selezione della velocità desiderata
- Selezione della direzione desiderata
- Avvio del movimento relativo rispetto alla posizione attuale
- Arresto del movimento
- Home Forward
- Home Reverse
- One step forward
- One step reverse
- Visualizzazione velocità massima
- Visualizzazione della velocità minima

All'interno della classe è stata implementata una *FSM* principale che descrive i diversi possibili stati e transizioni relativi al movimento, supportata da *FSM* secondarie dedicate alla gestione delle singole tipologie di movimento. Un esempio di tale struttura è riportato in figura 12.

```

Main FSM

Manual_Motor

if v_Enable=1 then
    case C_Manual of
        //=====
        Idle:
            if Comando_Motore.AxisError.HwError then
                v_ErrorState:=1;
                C_Manual:=Error;
            end_if;
            if (v_Vel<_V_MIN) | (v_Vel>_V_MAX) then
                v_Block:=TRUE;
            else
                v_Block:=FALSE;
            end_if;
            if v_Block=FALSE & v_Start=1 & v_Stop_movement=0 then
                C_Manual:=Movement;
                C_Movement:=10;
            end_if;
            if v_PowerOn=1 & PowerOff=0 then
                C_Manual:=M_PowerOn;
                C_PowerOn:=10;
            end_if;
            if v_PowerOn=0 & PowerOff=1 then
                C_Manual:=M_PowerOff;
                C_PowerOff:=10;
            end_if;
            if v_Stop_movement=1 then
                C_Manual:=M_Stop;
            end_if;

            if v_Block=FALSE & v_Home_forward=1 & v_Stop_movement=0 then
                C_Manual:=Home_forward;
                C_Home_forward:=10;
            end_if;
            if v_Block=FALSE & v_Home_reverse=1 & v_Stop_movement=0 then
                C_Manual:=Home_reverse;
                C_Home_reverse:=10;
            end_if;
            if v_Block=FALSE & v_one_Step_forward=1 & v_Stop_movement=0 then
                C_Manual:=OneStep_forward;
                C_OneStep_forward:=10;
            end_if;
            if v_Block=FALSE & v_one_Step_reverse=1 & v_Stop_movement=0 then
                C_Manual:=OneStep_reverse;
                C_OneStep_reverse:=10;
            end_if;
            if v_Reset=1 then
                C_Manual:=Reset;
            end_if;
        //=====
        M_PowerOn:
            case C_PowerOn of
                0:
                10:
                    if (Comando_Motore.AxisStatus.PowerOn=0) & (Comando_Motore.AxisStatus.ReadyToPowerOn) then
                        Comando_Motore.PowerOnMode:=LMCAXIS_MOVEDIRECTION::LMCAXIS_MOVE_ANY_WAY;
                    elseif Comando_Motore.AxisStatus.PowerOn then
                        C_PowerOn:=20;
                    end_if;
                20:
                    if Comando_Motore.AxisStatus.PowerOn=1 then
                        C_PowerOn:=0;
                        PowerOn.Write(input:=0);
                        C_Manual:=Idle;
                    end_if;
            end_case;
    end_case;

```

Secondary FSM

Figure 12: Esempio di FSM per la movimentazione manuale

Infine, è presente nella rete una classe denominata **AutoManSelector**, che consente l'abilitazione della rete di controllo manuale (*Manual Network*) oppure di quella automatica (*Automatic Network*), a seconda della modalità operativa selezionata dall'operatore. All'interno di questa classe è implementata anche la gestione dell'arresto di emergenza. Qualora l'operatore, tramite l'interfaccia grafica (GUI), impartisca un comando di arresto d'emergenza, viene inviato un segnale di stop e spegnimento a ciascun motore. In modo analogo, la stessa logica viene applicata nel caso di un comando di reset.

3.6 Axis Profiles network

All'interno di questa rete vengono implementate le classi che fungono da interfaccia tra il software e il modulo PLC ST-151, responsabile del controllo di ciascun motore.

Per ogni motore vengono utilizzate due classi predefinite, messe a disposizione direttamente dal modulo ST-151, ovvero:

- `_LMCAxis`
- `StepperControl`

Tali classi forniscono i metodi e le proprietà necessari per l'invio di comandi di movimento e la gestione dello stato del motore, consentendo una comunicazione diretta e strutturata con l'hardware.

`_LMCAxis`

La classe `_LMCAxis` rappresenta la classe base utilizzata per generare il profilo di movimentazione richiesto per un asse. Essa fornisce una serie di funzioni predefinite che possono essere utilizzate direttamente dall'utente, permettendo così di gestire in modo efficiente le traiettorie e i comandi di movimento.

Il comportamento della classe è configurabile attraverso alcuni parametri fondamentali, che devono essere impostati in base alla tipologia di controllo (rotativo o lineare) e alle specifiche del motore in uso. I principali parametri da configurare sono:

- **ExUnits** → rappresenta il numero di microstep moltiplicato per il numero di passi per giro:

$$\text{ExUnits} = \text{microsteps} \times \text{StepsPerRev} = 64 \times 400 = 25600$$

Dove `StepsPerRev` è un dato di targa del motore.

- **IntUnit** → rappresenta l'unità interna di misura:

- 360, se si tratta di un motore rotativo [gradi];
- lunghezza della corsa [millimetri], nel caso di motore lineare.

- **Vmax** → valore massimo della velocità, generalmente fornito come dato di targa del motore.

- **Amax** → valore massimo dell'accelerazione, tipicamente impostato come:

$$\text{Amax} = \text{Vmax} \times 10$$

`StepperControl`

La classe `StepperControl` rappresenta il vero e proprio controller del motore. Essa funge da interfaccia tra il generatore di profilo, ovvero la classe `_LMCAxis`, e la classe associata al modulo ST-151, responsabile della generazione del segnale fisico di comando.

In ingresso, `StepperControl` riceve il profilo di movimento generato da `_LMCAxis`; in uscita, fornisce il segnale di controllo effettivo per il motore, pilotandone l'esecuzione.

Anche questa classe necessita della configurazione di alcuni parametri fondamentali, tra cui:

- **StepAngle** → valore caratteristico del motore, fornito come dato di targa [gradi/step].
- **MaxVel** → velocità massima, anch'essa specificata come dato di targa [rpm].
- **MaxSpeed** → corrisponde al parametro `Vmax` della classe `_LMCAxis`, ma espresso in *application units*.

In figura 13 viene riportato il collegamento tra le classi `_LMCAxis` e `StepperControl`.

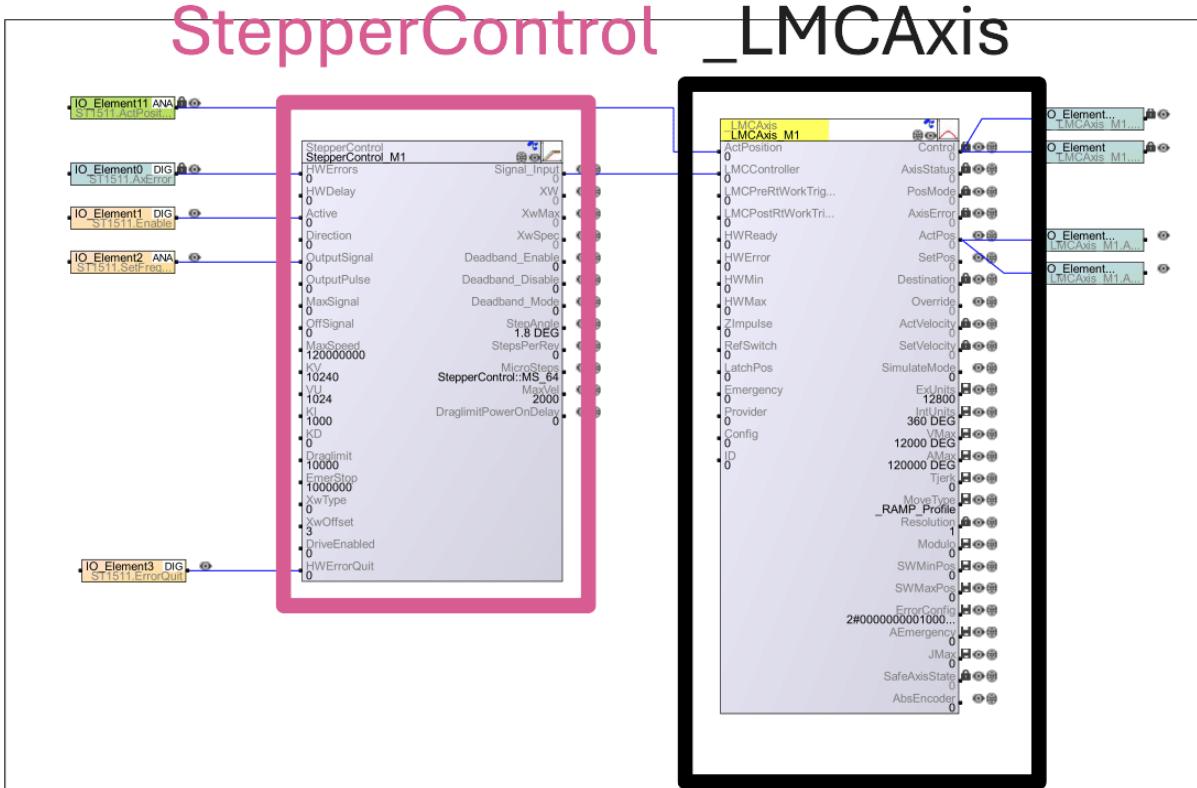


Figure 13: Profili di movimento generati e gestiti dal sistema

3.7 ModBus Communication network

All'interno di questa rete è stata implementata la comunicazione tra il PLC e il mondo esterno. Il protocollo di comunicazione utilizzato è **ModBus TCP/IP**, basato su tecnologia Ethernet. La scelta di questo protocollo è stata dettata dalla semplicità di implementazione sia lato PLC sia lato EPICS.

L'implementazione di questa rete si è resa necessaria in quanto, parallelamente allo sviluppo del software di controllo, è stata realizzata un'interfaccia grafica basata su *Control System Studio (CS-Studio)*. Quest'interfaccia consente all'operatore di comandare e monitorare l'esperimento **IRIS**.

Il protocollo ModBus è un protocollo di tipo **client-server**. In questo contesto:

- Il **client** (o *master*) è il dispositivo che inizia la comunicazione, inviando richieste per leggere o scrivere dati nei registri del server;
- Il **server** (o *slave*) è il dispositivo che risponde alle richieste del client.

Nel nostro caso, poiché il PLC deve fornire dati e ricevere comandi, esso assume il ruolo di **server ModBus**, mentre l'**IOC EPICS** agisce come **client**.

I dati condivisi tramite questa comunicazione riguardano sia la movimentazione automatica che quella manuale. Una lista dettagliata delle variabili condivise è riportata in **appendice**.

Per la gestione del protocollo sono state utilizzate tre differenti istanze della classe **_ModBusTCP_SLAVE**, una classe predefinita messa a disposizione dal sistema e configurabile secondo le necessità; può essere importata dalla libreria **Tools>Sigmatek>Tools>Communications**.

Tra i parametri più rilevanti troviamo:

- **cPort** → identifica la porta TCP utilizzata per la comunicazione ModBus;
- **cTable** → collegamento a una classe contenente le variabili da condividere, organizzate sotto forma di "Tables". Nel caso in cui si vogliano condividere un set di variabili attraverso delle *Tables* è necessario inizializzarlo con **2#11**

- to_AsciiBin: client da collegare.

Per garantire una struttura ordinata e modulare, è stata adottata una suddivisione delle variabili condivise in tre macro-sezioni, ognuna gestita da un'istanza distinta di `_ModBusTCPSLAVE`:

- **Motion control - manual - motors WRITE**
- **Motion control - manual - motors READ**
- **Others:**
 - Motion control - manual - solenoid
 - Motion control - automatic - main control
 - Correction steps - manual
 - Motors status

Esempio di condivisione variabili ModBus

Per implementare le variabili da condividere è necessario la creazione di una classe senza definire la voce "Task Settings"; tasto destro sulla classe>New Table. A questo punto bisogna creare un metodo "Read" nel server "ClassSvr" e definire questo:

```
1 output := (#NomeTable()\$DINT)
```

. Inoltre, per il corretto funzionamento del server, è necessario utilizzare dei metodi della classe `_ModBusTCPSLAVE` in un determinato ordine. E' stato quindi implementata la classe `ControlloServerModbus` da collegare alla classe `_ModBusTCPSLAVE` nella quale vengono implementate queste righe di codice:

```
1 if _FirstScan then
2   oc_ModBusTCPSlave.sDisable:=1; //servere deve essere inizialmente disabilitato
3   oc_ModBusTCPSlave.Init();
4 end_if;
5
6 Tempo:=Tempo+100;
7
8 if Tempo=2000 then
9   oc_ModBusTCPSlave.sDisable:=0; //dopo essere stato inizializzato puo' essere abilitato
10 end_if;
11
12 oc_ModBusTCPSlave.CyclicCall();
```

In figura 14 viene riportato un esempio di implementazione della condivisione di alcune variabili appartenenti alla macro-sezione **Motion control - manual - motors WRITE**.

Nella parte superiore della figura è illustrata la configurazione dettagliata per la condivisione di una singola variabile, evidenziando il modo in cui questa viene mappata all'interno della tabella ModBus.

La formattazione per la corretta condivisione di una variabile (server di una classe) deve essere la seguente:

- MODBUS address.
- Number of MODBUS registers (1 o 2 registri ovvero 16-bit o 32-bit).
- Access rights (0→Read; 1→Write; 2→Read and Write).

Gestione dispositivi ausiliari: Rack12V e MPS

Oltre alla gestione della movimentazione, nella rete sono presenti ulteriori classi dedicate al controllo di dispositivi ausiliari:

- **Rack12V**

Questa classe implementa la logica di accensione e spegnimento della *rack-box* che alimenta i dispositivi a 12V, tra cui il LED interno al sistema IRIS.

- **MPS (Machine Protection System)**

La classe MPS gestisce la comunicazione con il *Machine Protection System - MPS*. In particolare, è responsabile della trasmissione del segnale di richiesta di disaccoppiamento della camera, nonché della ricezione dell'*acknowledgement* che dà il via libera al disaccoppiamento della camera.

HoldingRegister

```

FUNCTION TAB HoldingRegister::TableHR
//-----MODBUS address | number of MODBUS registers | access rights | server name
//-----Motion control - manual - motors
WRITE-----
10$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.PowerOn",
12$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.PowerOff",
14$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.Vel",
16$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.Start",
18$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.Stop_Movement",
20$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.Direction",
22$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.S_Reset",
24$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.Pos",
26$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.s_Home_forward",
28$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.s_Home_reverse",
30$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.s_one_step_reverse",
32$UINT, 2$USINT, 2$USINT, "M1_Charge_Slider.s_one_step_forward",
34$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.PowerOn",
36$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.PowerOff",
38$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.Vel",
40$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.Start",
42$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.Stop_Movement",
44$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.Direction",
46$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.S_Reset",
48$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.Pos",
50$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.s_Home_forward",
52$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.s_Home_reverse",
54$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.s_one_step_reverse",
56$UINT, 2$USINT, 2$USINT, "M2_Charge_Buffer.s_one_step_forward",

```

Figure 14: Esempio di configurazione variabili condivise – Motion control (manual motors - WRITE)

4 Descrizione classi singoli motori

4.1 Motore M1 - Charge slider motor

Questo motore è adibito alla movimentazione del componente denominato *Charge Slider*. Esso ha due posizioni possibili; accoppiato e disaccoppiato. Alle funzioni base vengono aggiunte le seguenti funzioni:

- GetCoupledState
- GetMoveDone

"GetCoupledState", utilizzata dalla classe **Manager Motori**, fornisce lo stato di accoppiamento o disaccoppiamento del *Charge Slider*. *"GetMoveDone"*, anch'essa utilizzata dal **Manager Motori**, gli fornisce lo stato di compimento dello *Swing Movement* richiesto.

Descrizione dello stato **Movement**:

- Le movimentazioni possibili sono due:

Type 0: Disaccoppiato → Accoppiato

Type 1: Accoppiato → Disaccoppiato

Type 2: Swing Movement → doppio movimento relativo alla posizione di accoppiamento per evitare eventuali blocchi del target durante il percorso.

4.2 Motore M2 - Charge buffer motor

Il motore in oggetto è destinato al trasferimento di pellet dallo scivolo di carico iniziale al charge slider. I pellet, introdotti nello scivolo iniziale, cadono per gravità fino a raggiungere l'interfaccia di ingresso del charge buffer, dove vengono incanalati uno alla volta in uno dei due slot disponibili. Poiché la fase di carico risulta particolarmente delicata sotto il profilo meccanico e operativo, richiedendo il coinvolgimento simultaneo di più motori e dovendo rispettare specifici vincoli meccanici, è necessario che il motore M2 operi in coordinamento con il motore del Central Movement (M5) e con il charge slider (M1). Per fare ciò vengono implementate queste funzioni:

- ReadM5Position
- GetPelletInSlider

- Req_M1_Mov
- ACK_M1_Mov

Le funzioni implementate hanno lo scopo di interfacciarsi con il **Manager Motori**, consentendo lo scambio di informazioni relative allo stato del sistema di movimentazione.

Attraverso la funzione **ReadM5Position**, il motore M2 verifica lo stato del motore M5, questo gli viene comunicato dal **Manager Motori**, questa informazione indica se M5 si trova nella posizione corretta per consentire il carico di un pellet.

La funzione **GetPelletInSlider** permette invece a M2 di comunicare al **Manager Motori** il proprio stato, indicando se un pellet è stato correttamente rilasciato all'interfaccia con M5.

Infine, le ultime due funzioni sono dedicate alla richiesta e al monitoraggio del movimento di tipo *Swing Movement* eseguito dal motore M1.

Descrizione dello stato **Movement**:

- La movimentazione è strettamente vincolata alla progettazione meccanica del *Charge Buffer*, il quale è dotato di due slot per il caricamento dei pellet e di un finecorsa (Limit Switch, LS). A partire dalla posizione di homing, un pellet viene caricato per gravità nello slot "1". Successivamente, il *Charge Buffer* effettua una rotazione completa in un determinato verso: durante questa rotazione, un secondo pellet viene caricato nello slot "2" fino al raggiungimento del finecorsa, mentre il pellet inizialmente presente nello slot "1" viene scaricato nel *Charge Slider* e trasportato fino all'interfaccia con M5. Il caricamento del terzo pellet, quando abilitato, avviene mediante una rotazione del *Charge Buffer* in senso opposto. Durante tale movimento, il pellet presente nello slot "2" viene scaricato, mentre un nuovo pellet viene caricato nello slot "1". La movimentazione del *Charge Buffer* avviene quindi in modo alternato nei due versi di rotazione, in funzione del caricamento dei pellet. Questa alternanza è necessaria per via della conformazione fisica del dispositivo e viene mantenuta anche tra cicli di lavoro consecutivi, al fine di evitare la dispersione dei pellet.

La parte di movimentazione si distingue in "1 pellet mode" oppure "3 pellet mode" a seconda della scelta effettuata dall'operatore. Nel caso in cui venga scelto "1 pellet" le operazioni di carica eseguite saranno solo per 1 pellet. Riportiamo una parte di codice della macchina a stati che comanda la movimentazione:

```

1    case Movement_step of
2
3      0:
4        Movement_step:=v_StartMovementCase;
5
6      10:
7        if (Comando_M2.AxisStatus.PowerOn=0) & (Comando_M2.AxisStatus.ReadyToPowerOn) then
8          Comando_M2.PowerOn(Mode:=LMCAxis_MOVE_ANY_WAY);
9        elseif Comando_M2.AxisStatus.PowerOn then
10          if v_AlternatingHoming=1 then
11            Movement_step:=20;
12          elseif v_AlternatingHoming=-1 then
13            Movement_step:=200;
14          end_if;
15          end_if;
16
17      20:
18        Comando_M2.MoveRelative(Position:=300000, Mode:=LMCAxis_MOVE_RELATIVE_TO_POSITION, Speed:=V_M2, Accel:=A_M2, Decel:=A_M2, Jerk
19          :=J_M2);
20        Movement_step:=30;
21
22      30:
23        if Comando_M2.AxisStatus.InPosition & Comando_M2.AxisStatus.Standstill then
24          Comando_M2.MoveAbsolute(Position:= 2300000, Speed:=V_M2, Accel:=A_M2, Decel:=A_M2, Jerk:=J_M2);
25          s_pelletInIrisCharge.Write(input:=1);
26          v_InCharging+=1;
27          s_tablet_InCharging.Write(input:=v_InCharging);
28          Movement_step:=40;
29          end_if;
30
31      40:
32        if M2_LS.Read() | Comando_M2.AxisStatus.InPosition then
33          v_target_in_charge_slider:=1;
34          Comando_M2.StopMove(Decel:= A_M2, Jerk:= J_M2);
35          Movement_step:=50;
36          end_if;
37
38      50:
39        if M5_InPos then
40          V_CONTATORE_TARGET_CARICATI+=1;
41          v_Req_No_Stuck_Mov_M1:=1;
42          v_target_in_charge_slider:=0;
43          Movement_step:=60;
44          end_if;
45
46      60:
47        if v_ACK_M1_No_Stuck_Mov then
48          v_Req_No_Stuck_Mov_M1:=0;
49          v_ACK_M1_No_Stuck_Mov:=0;
50          v_InCharging+=1;
51          s_pelletInIrisCharge.Write(input:=2);
52          s_tablet_InCharging.Write(input:=v_InCharging);
53          if s_1_3_pelletMode.Read()=1 then //1 pellet

```

```

53     Movement_step:=180;
54     elseif s_1_3_pelletMode.Read()=0 then
55         Movement_step:=70;
56     end_if;
57     end_if;

```

Nello stato 10 M2 viene acceso. Lo stato 20 è necessario in quanto il motore si trova nella posizione di *Homing*, condizione in cui viene attivato il finecorsa. Per consentire un successivo movimento, è indispensabile allontanare il motore da tale posizione tramite un *movimento relativo*. Nello stato 30, il comando **MoveAbsolute** viene utilizzato come se fosse di tipo *endless*, ovvero il motore si muove fino a che non raggiunge il finecorsa (LS). Nello stato 40 quando il finecorsa viene attivato, M2 viene fermato e viene segnalato che un pellet è presente all'interfaccia con M5. Nello stato 50 si attende che M5 abbia finito la sua rotazione e che quindi abbia caricato un pellet uno dei tre slot disponibili; una volta terminata la rotazione viene richiesto lo *Swing movement*. Nello stato 60 si attende il completamento dello *Swing movement* e una volta terminata la FSM viene indirizzata a caricare altri pellet o a terminare la procedura di carica.

4.3 Motore M3 - Discharge Slider

Questo motore è adibito alla movimentazione del componente "discharge slider". Viene modellato in maniera del tutto analoga al motore M1 relativo al *Charge Slider*. Non viene però implementato lo "Swing movement"

4.4 Motore M4 - Discahrge Buffer

Questo motore è responsabile della movimentazione del componente denominato **Discharge Buffer**. Attraverso tale meccanismo, i pellet vengono trasferiti dal *Discharge Slider* a una zona dedicata alla misurazione, nella quale operano specifici rivelatori. In quest'area, il pellet viene posizionato correttamente e mantenuto in posizione tramite un motore dedicato fino al termine della misurazione. Per garantire la corretta esecuzione di questa fase, il motore del *Discharge Buffer* opera in stretta sinergia con i motori **M5** e **M8**.

A tal fine, sono implementate le seguenti funzioni:

1. **GetMeasurementStartOver**
2. **SetDischargeOver**
3. **GetM7Status**
4. **GetMoveDone**
5. **GetPelletAvailable**
6. **GetErrorState**

Le funzioni 1, 2, 4, 5 e 6 vengono utilizzate dalla classe **Manager Motori** per aggiornare lo stato relativo, rispettivamente allo stato di start/stop misura comandato dall'utente, leggere se la fase di scarica, e quindi di misurazione è completata, per informare se la movimentazione richiesta è terminata, per informare se la classe M4 è entrata nello stato di errore ed infine segnalare quanti pellet sono stati caricati dall'utente.

La macchina a stati del motore associato al modulo **ST-151** è progettata per gestire le diverse fasi operative relative alla movimentazione dei pellet, in particolare dallo *discharge buffer* alla posizione di misura *offline*. Di seguito si riportano gli stati principali implementati:

- **Movement**: gestisce la movimentazione dei pellet dal *discharge slider/buffer* alla postazione di misura *offline*. In questa fase è fondamentale la coordinazione con il motore **M7**. Come nel caso del motore **M2**, anche qui la logica della macchina a stati si biforca in base alla modalità operativa: *1 pellet* oppure *3 pellet*.
- **Movement 2**: questa fase di movimentazione viene attivata durante la fase di *discharging*. In una prima parte è prevista l'interazione con il motore **M5**, poiché i tre pellet vengono scaricati dal sistema *Central Movement*. Per evitare il rischio di blocco all'interno del *discharge slider*, non si

procede con lo scarico di tutti e tre i pellet, ma ciascun pellet viene trasferito in uno degli slot del *discharge buffer*. Questa operazione richiede una coordinazione tra i motori **M5** e **M4**, ottenuta tramite la funzione **GetMoveDone**.

Oltre alla gestione degli stati, questa classe si occupa dell'avvio e dell'arresto di timer che misurano:

- Il tempo trascorso tra la fine dell'irraggiamento e l'inizio della misura *offline*;
- La durata effettiva della fase di misura.

Queste funzionalità sono implementate tramite i client **c_TimerDis** e **c_TimerMov**, che interagiscono rispettivamente con le classi **DischargeTimer1** e **MoveTimer1** nel network Timer. In figura 15 viene riportato il codice relativo a **Movement**.

```

Motore_M4

Cmd_motore_M4_movement:

if Comando_M4.AxisError.HwError then
    v_ErrorState:=1;
    Movement_step:=0;
    MotoreM4Step:=Cmd_motore_M4_Error;
end_if;

case Movement_step of
0:
    Movement_step:=v_StartMovementCase;
10:
    if (Comando_M4.AxisStatus.PowerOn=0) & (Comando_M4.AxisStatus.ReadyToPowerOn) then
        Comando_M4.PowerOn(Mode:=LMCAxis_MOVE_ANY_WAY);
    elseif Comando_M4.AxisStatus.PowerOn then
        Movement_step:=20;
        s_DischargeStatus.Write(input:=0);
    end_if;
20:
    if v_M7_inBlockPos then
        Movement_Step:=30;
    elseif v_M7_inBlockPos=FALSE then
        Comando_to_M7.SetSequenceState(Cmd_motore:=Motore_M7::Cmd_motore_M7_movement, Motion_type:=0, StartCase:=10); //dico al motore di andare in posizione di
    end_if;
25:
    if v_M7_inBlockPos=true then
        Movement_Step:=30;
        flag:=1;
    end_if;
30:
    Comando_M4.MoveAbsolute(Position:=(P1_M4+v_offset_Dis), Speed:=V_M4, Accel:=A_M4, Decel:=A_M4, Jerk:=J_M4);
    Movement_Step:=40;
40:
    IF Comando_M4.InPositionMode:=LMCAxis_NO_POSITIONWINDOW, Positionwindow:=0) & Comando_M4.AxisStatus.Standstill & v_MeasurementStartOver THEN //se ho
scaricato target e dico che è pronto per la misurazione
    //FASE DI MISURAZIONE
    v_pelletMeasured:=1;
    s_pelletMeasured.write(input:=v_pelletMeasured);
    c_TimerDis.write(input:=0);
    c_TimerMov.write(input:=0);
    s_measurementON.write(input:=1); //MISURAZIONE ATTIVA
    s_DischargeStatus.write(input:=1);
    v_target_misurati+=1;
    Movement_step:=50;
END_IF;
50:
    if v_MeasurementStartOver=FALSE then
        v_pelletMeasured-=1;
        s_pelletMeasured.write(input:=v_pelletMeasured);
        c_TimerDis.write(input:=0);
        Comando_to_M7.SetSequenceState(Cmd_motore:=Motore_M7::Cmd_motore_M7_movement, Motion_type:=1, StartCase:=10);
        v_target_misurati+=1;
        v_pelletInTrolley+=1;
        s_pelletInTrolley.write(input:=v_pelletInTrolley);
        s_measurement_ON.write(input:=0);
        v_offset_Dis+=1200000;
    if s_1_3_pelletMode.Read()=1 then
        Movement_Step:=80;
    elseif s_1_3_pelletMode.Read()=0 then
        Movement_Step:=60;
    end_if;
    end_if;
60:
    if v_target_misurati<6 & Comando_to_M7.GetCoupledState()=FALSE then
        Movement_Step:=20;
    elseif v_target_misurati=6 & Comando_to_M7.GetCoupledState()=FALSE then
        Comando_M4.PowerOff(Mode:=LMCAxis_IMMEDIATE_STOPP);
        Movement_Step:=70;
    end_if;
70:
    if Comando_M4.AxisStatus.PowerOn=0 then
        v_DischargeMeasure_End:=TRUE;
        Movement_Step:=0;
        MotoreM4Step:=Cmd_motore_M4_idle;
    end_if;
80:
    if Comando_to_M7.GetCoupledState()=FALSE & Comando_M4.AxisStatus.Standstill then
        s_DischargeStatus.write(input:=2);
        Comando_M4.Poweroff(Mode:=LMCAxis_IMMEDIATE_STOPP);
        Movement_Step:=70;
    end_if;
end_case;

```

Figure 15: Movement M4

Lo stato 10 accende il motore. Lo stato 20 controlla se M7 è in posizione di blocco e se questo non lo è lo comanda per andarci. Raggiunto tale posizione, nello stato 30 viene comandato di ruotare fino ad una determinata posizione, ruotando un pellet viene scaricato nella guida di plastica fino al punto di

detection. Nello stato 40 è presente una struttura *if* nella quale si accede se M4 è fermo e se arriva il comando di start misurazione dall'utente. All'interno della struttura:

- Aggiornato contatori per monitorare i pellet processati.
- Aggiornamento di server per la GUI.
- Avvio timer riguardante il tempo di misura.
- Stop timer riguardante tempo di movimentazione.

Lo stato 50 è deputato all'attesa del comando di arresto. All'interno della struttura *if*, vengono aggiornati contatori e server, viene arrestato il timer relativo al tempo di misurazione, e viene inviato al motore M7 il comando per uscire dalla posizione di blocco pastiglia. Successivamente, viene aggiornato un offset che sarà utilizzato per determinare una nuova posizione di destinazione per il motore M4. In questo stato, inoltre, viene instradata la macchina *FSM* verso il caricamento di uno o tre pellet, a seconda delle condizioni operative selezionate dall'utente. Nello stato 60, invece, viene verificato il numero di pellet già caricati.

4.5 Motore M5 - Central Movement

Questo è il motore più complesso da comandare poichè risulta essere attivo nelle fasi di charging, irradiation e discharging co-operando dunque con diversi motori. Questo motore è adibito alla movimentazione del componente denominato "Central movement". Vengono implementate le seguenti funzioni:

1. GetPositionChargeState
2. GetChargingStatus
3. GetIrradiationStatus
4. GetDischargingStatus
5. GetIrradiationStartOver
6. Req_M3_Mov
7. ACK_M3_Mov
8. Req_M4_Mov
9. ACK_M4_Mov
10. ACK_M1_Mov

Attraverso la funzione 1, M5 fornisce al **Manager Motori** l'informazione relativa al corretto posizionamento di M5 per il caricamento di un pellet. La funzione 2, 3 e 4 permettono di informare il **Manager Motori** che le fasi di Charge, irradiation e discarghe sono terminate. La funzione 5 invece viene utilizzata dal **Manager_globale** per aggiornare lo stato del comando start/stop irraggiamento inviato dall'utente. Le restanti funzioni servono per richiedere e monitorare dei movimenti richiesti ai motori M1, M3 e M4. Infine M5 comanderà direttamente i due attuatori utilizzati per bloccare gli ingressi e le uscite degli slot di caricamento nel central movement, definiti dalla classe Attuatore.

Di seguito la spiegazione delle tre tipologie di movimentazione richieste a M5:

- **Movement_CHARGING:** A partire dalla posizione di *Homing*, l'attuatore 1 viene estratto e rimane in attesa fino a quando il **Manager Motori** — attraverso il motore **M2** — segnala la presenza di un pellet pronto per il caricamento. Una volta ricevuta la conferma, viene inviato un comando per ruotare **M5** nella posizione di *carica*. Durante la rotazione del motore, il perno dell'attuatore 1 interagisce meccanicamente con uno sportellino presente su ogni slot di M5, aprendolo e permettendo l'accesso ai pellet. Al termine del caricamento, viene richiesto lo *Swing Movement* al motore **M1**. Una volta completato tale movimento, l'attuatore 1 viene retratto, permettendo la chiusura dello sportellino tramite una molla di richiamo. Questo processo viene ripetuto nel caso in cui sia stata selezionata la modalità operativa "**3 pellet**".

- **MovementIRRADIATION:** La fase di irraggiamento prevede di portare nella posizione di irraggiamento i 3 pellet uno alla volta. Lo spostamento e la permanenza in tale posizione può essere automatizzata o manuale, ma comunque gestita da una classe esterna. L'operatore può decidere se interagire attraverso una modalità "base" o "avanzata". Nella prima l'operatore decide esclusivamente il tempo di permanenza dei vari pellet nella posizione di irraggiamento e comanda l'inizio dell'intera procedura. Nella modalità "avanzata" invece è necessario comandare l'accoppiamento/disaccoppiamento della camera ed inoltre è l'operatore che attraverso i comandi di start/stop irraggiamento decide il tempo di permanenza di ciascun pellet nella posizione di irraggiamento.
- **MovementDISCHRGING:** La fase di scarica prevede di scaricare i pellet nel discharge slider e successivamente portarli nel discharge buffer. Per scaricare il primo pellet irraggiato si porta nuovamente M5 nello zero, si porta in estrazione l'attuatore 2 che permette di aprire lo sportello degli slot durante la rotazione di M5. Ad ogni pellet scaricato, si richiede lo "Swing Moveemnt" per il discharge slider e al suo termine si richiede anche al charge buffer, M4, di ruotare a caricare un pellet in modo da liberare lo scivolo e evitare quindi il sovrapporsi di pellet. Per la scarica degli altri pellet vengono ripetute le medesime operazione, nel caso in cui si operi in modalità "3 pellet".

In figura 16 è riportata una porzione del codice utilizzato per gestire la fase di carica del motore M5. Nel stato 5, il motore M5 viene attivato e l'attuatore 1 viene portato nella posizione estratta. Nello stato 10, il sistema attende che il pellet sia presente all'interfaccia di M5; tale informazione è fornita dal Manager Motori tramite la variabile `v_TargetAvailableM2`. Se il pellet risulta disponibile, a M5 viene comandata la rotazione fino a una posizione predefinita, in modo da permettere l'inserimento del pellet nello slot. Raggiunta tale posizione (stato 20), vengono aggiornati contatori e server. La richiesta di *swing movement* viene gestita direttamente dal motore M2; di conseguenza, M5 riceve soltanto l'informazione relativa all'avvenuto completamento del movimento. Una volta ricevuta questa conferma, si accede allo stato 30, in cui l'attuatore 1 viene riportato in posizione retratta. In questo stato si determina inoltre se è necessario procedere con il caricamento dei due pellet rimanenti oppure concludere la fase di carica.

Le parti di codice relative all'irraggiamento e alla scarica sono state sviluppate con una logica del tutto simile a quella utilizzata per la parte di carica.

Motore_M5

```

Cmd_motore_M5_movement_CHARGING:

if Comando_M5.AxisError.HwError then
    v_ErrorState:=1;
    v_MovementChargeStep:=0;
    MotoreM5Step:=Cmd_motore_M5_Error;
end_if;

case v_MovementChargeStep of
0://idle
    v_MovementChargeStep:=v_StartMovementChargingCase;
    trigger_1:=1;

5:
    c_MaxCurrent.Write(input:=MOTORMAXCURRENT);
    if (Comando_M5.AxisStatus.PowerOn=0) & (Comando_M5.AxisStatus.ReadyToPowerOn) then
        Comando_M5.PowerOn_Mode:=LMCAXIS_MOVEDIRECTION::LMCAXIS_MOVE_ANY WAY); //può ruotare in entrambi i versi
    elsif Comando_M5.AxisStatus.PowerOn then
        Comando_Attuatore_1.SetSequenceState(Param:=Attuatore::Push); //to pushed position
        v_M5_charge_position:=false;
        v_MovementChargeStep:=10;
    end_if;

10:
    if Comando_Attuatore_1.GetActuatorStatus()=1 & v_TargetAvaiableM2=1 then
        Comando_M5.MoveAbsolute(Position:=P1_M5+v_ccs1, Speed:=v_M5, Accel:=A_M5, Decel:=A_M5, Jerk:=J_M5);
        v_MovementChargeStep:=20;
    end_if;

20:
    if Comando_M5.AxisStatus.InPosition=1 & Comando_M5.AxisStatus.Standstill then
        v_Tablet_caricati+=1;
        v_pelletRemaining-=1;
        s_PelletRemaining.Write(input:=v_pelletRemaining);
        v_M5_charge_position:=true;
        v_MovementChargeStep:=25;
    end_if;

25:
    if v_ACK_M1_No_Stuck_Mov then
        v_ACK_M1_No_Stuck_Mov:=0;
        v_M5_charge_position:=true;
        v_MovementChargeStep:=30;
    end_if;

30:
    v_M5_charge_position:=FALSE;
    Comando_Attuatore_1.SetSequenceState(Param:=Attuatore::Pull); //to pulled position
    if s_1_3_pelletMode.Read()=1 then //1 pellet
        v_MovementChargeStep:=110;
    elsif s_1_3_pelletMode.Read()=0 then
        v_MovementChargeStep:=35;
    end_if;

35:
    if v_TargetAvaiableM2=1 then
        v_MovementChargeStep:=50;
    end_if;

50:
    if v_TargetAvaiableM2=1 then //pushed
        Comando_Attuatore_1.SetSequenceState(Param:=Attuatore::Push);
        Comando_M5.MoveAbsolute(Position:=P2_M5+v_ccs2, Speed:=0.7*v_M5, Accel:=0.7*A_M5, Decel:=0.7*A_M5,
        Jerk:=0.7*J_M5);
        v_MovementChargeStep:=60;
    end_if;

```

Figure 16: Codice parziale Movement_CHARGING

4.6 Motore M6 - Allineamento camera

Questo motore è adibito alla movimentazione della camera nella quale avverrà il vuoto e che ospita il central movement. Questo motore accoppia e disaccoppia la camera da vuoto. Viene implementata la funzione:

- GetCoupledState

Attraverso questa il Manager Motori è in grado di stabilire se la camera è accoppiata o meno.

4.7 Attuatore 1 & 2

All'interno di questa classe viene definita la seguente macchina a stati:

- **Push:** Porta il perno in posizione estratta.
- **Pull:** Porta il perno in posizione retratta.

Questa attraverso gli stati di Push e Pull comanda in maniera opportuna le uscite digitali del modulo TO-127 attraverso le quali vengono comandati le bobine di controllo dei diversi relè che forniscono il gradino di tensione positiva o negativa ai solenoidi contenuti all'interno degli attuatori.

4.8 STO - Arresto di emergenza

Lo stop di emergenza è uno stop che può essere azionato attraverso un apposito pulsante di emergenza presente sul front panel della motors box oppure via software attraverso un pulsante presente sulla GUI. Quando viene azionato il pulsante fisico questo chiude due circuiti distinti i quali sono collegati alle entrate "Enable 1" e "Enable 2" di ogni modulo ST-151. Quando uno di questi due ingressi non viene visto presente il modulo PLC scollega la parte di potenza in maniera automatica e fornisce in uscita un errore relativo all'arresto di emergenza.

Dal punto di vista software è stato implementato una funzione simile, ovvero quando premuto le varie FSM implementate entrano in uno stato di **stop** e ad ogni motore viene comandato di fermarsi e viene spento, rimanendo in attesa di un reset. Premendo il pulsante fisico o virtuale di reset, vengono cancellati gli errori presenti e riportata la macchina ad uno stato iniziale nel quale è necessario ricominciare dalla procedura di homing. Nello stato di emergenza è possibile anche entrare quando viene rilevato un errore a livello dal modulo PLC. Per implementare questa funzione, in ogni stato della macchina a stato interna di ogni motore vengono utilizzate le seguenti linee di codice:

```
1     if Comando_Mx.AxisError.HwError then
2         v_ErrorState:=1;
3         MovemenChargeStep:=0;
4         MotoreMxStep:=Cmd_motore_Mx_Error;
5     end_if;
```

Queste linee di codice vengono implementate in ogni stato delle FSM prima di eseguire il codice relativo alla struttura *Case*. In questo modo questa parte del codice viene eseguita ad ogni ciclo macchina monitorando gli eventuali stati di errori che possono verificarsi. Se un errore viene rilevato ogni FSM viene portata nello stato di errore, stop e stopped. Al contempo la variabile *v_ErrorState* viene portata ad un valore alto; questa variabile viene letta dalla classe **Manager Motori**, come nel codice riportato sotto, e porta tale classe nello stato di errore ed innesca un sistema a catena che porta anche le classi **Main Sequence** e **Manager globale** in uno stato di errore.

```
1     if M1_status.GetErrorHandler()=1 | M2_status.GetErrorHandler()=1 | M3_status.GetErrorHandler()=1 | M4_status.GetErrorHandler()=1 |
2         M5_status.GetErrorHandler()=1 | M6_status.GetErrorHandler()=1 then
3             ManagerMotorStep:=Cmd_ManagerMotori_stop;
4             v_ErrorState:=1;
5             Charge_step:=0;
6         end_if;
```

Nella classe **Manager Motori**, nel caso in cui uno dei motori entri uno stato di errore anche tutti gli altri motori vengono stoppati per evitare movimentazioni non previste ed eventuali collisioni. Quando lo stato di **stop** viene abilitato viene utilizzato il seguente codice:

```
1     Cmd_ManagerMotori_stop:
2
3         M1_command.SetSequenceState(Cmd_motore:=cmd_motore_M1_stop, Motion_type:=0, StartCase:=0);
4         M2_command.SetSequenceState(Cmd_motore:=cmd_motore_M2_stop, StartCase:=0);
5         M3_command.SetSequenceState(Cmd_motore:=cmd_motore_M3_stop, Motion_type:=0, StartCase:=0);
6         M4_command.SetSequenceState(Cmd_motore:=cmd_motore_M4_stop, Motion_type:=0, StartCase:=0);
7         M5_command.SetSequenceState(Cmd_motore:=cmd_motore_M5_stop, Motion_type:=0, StartCase:=0);
8         M6_command.SetSequenceState(Cmd_motore:=cmd_motore_M6_stop, Motion_type:=0, StartCase:=0);
9
10        ManagerMotorStep:=Cmd_ManagerMotori_Stopped;
11
12        Stop_client.SetSequenceState(Cmd:=MainSequence::Cmd_stop);
13        Stop_client.SetSequenceState_2(Param:=MainSequence::Stop);
```

da questo stato, una volta sbloccato il pulsante d'emergenza (se premuto) e' possibile avviare il reset, tale procedura prevede di comandare le varie FSM dei motori di entrare nello stato di reset:

```

1   Cmd_ManagerMotori_Reset:
2
3     M1_command.SetSequenceState(Cmd_motore:=Motore_M1::Cmd_motore_M1_reset, Motion_type:=0, StartCase:=0);
4     M2_command.SetSequenceState(Cmd_motore:=Motore_M2::Cmd_motore_M2_reset, StartCase:=0);
5     M3_command.SetSequenceState(Cmd_motore:=Motore_M3::Cmd_motore_M3_reset, Motion_type:=0, StartCase:=0);
6     M4_command.SetSequenceState(Cmd_motore:=Motore_M4::Cmd_motore_M4_reset, Motion_type:=0, StartCase:=0);
7     M5_command.SetSequenceState(Cmd_motore:=Motore_M5::Cmd_motore_M5_reset, Motion_type:=0, StartCase:=0);
8     M6_command.SetSequenceState(Cmd_motore:=Motore_M6::Cmd_motore_M6_reset, Motion_type:=0, StartCase:=0);
9
10    ManagerMotoriStep:=Cmd_ManagerMotori_Idle;
11    .
12    .
13    .

```

Mentre il codice utilizzato per il reset del singolo motore è:

```

1   Cmd_motore_Mx_Reset:
2
3     Comando_M1.QuitError(); //reset degli errori nel modulo ST-151
4     Mx_homed:=FALSE;
5     Mx_phased:=FALSE;
6     v_ErrorState:=0;
7     if Comando_Mx.AxisStatus.FiltRdy then
8       MotoreMxStep:=Cmd_motore_Mx_Idle;
9     end_if;

```